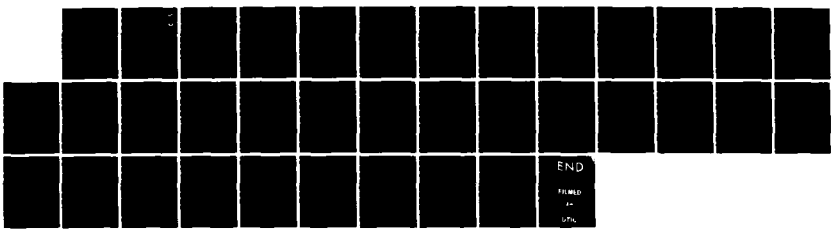
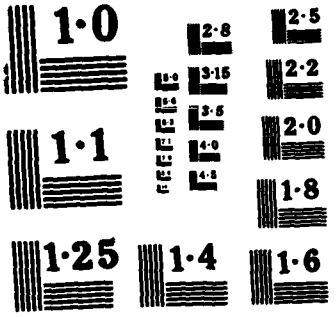
AD-A163 426 SPACE-EFFICIENT ALGORITHMS FOR COMPUTATIONAL GEOMETRY 1/1
(U) MASSACHUSETTS INST OF TECH CAMBRIDGE DEPT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNCLASSIFIED C A PHILLIPS OCT 85 N00014-80-C-0622 F/G 9/2 NL





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS 02139

VLSI Memo No. 85-270 October 1985

Space-Efficient Algorithms for Computational Geometry*

Cynthia A. Phillips**

ABSTRACT

This thesis presents an algorithm for determining the connectivity of a set of N rectangles in the plane, a problem central to avoiding aliasing in VLSI design rule checkers. Previous algorithms for this problem either worked slowly with a small amount of primary memory space, or worked quickly but used more space. The algorithm presented here, based upon a technique called scanning, operates in O(N lg N) time in the worst case. This matches the running time of the best known sequential algorithm for this problem. Because we use a machine model that explicitly incorporates secondary memory, the new connected components algorithm avoids unexpected disk thrashing which leads to lower performance. The algorithm uses O(W) primary memory space, where W, the scan width, is the maximum number of rectangles to cross any vertical cut. It requires no more than O(N) transfers between primary and secondary memory.

When a vertical line passes through a set of rectangles, those rectangles cut by the line form a set of line segments. The key to development of space-efficient algorithms using a two layer memory model is that appropriate manipulations of these segments alone can solve more complicated problems such as the connected components problem. This thesis introduces interval trees, a simple, sparse, data structure for storing a set of k line segments. With this data structure, a variation on a balanced search tree, one can perform each of the following operations in O(lg k) time in the worst case: 1) insert a new segment, 2) delete a segment, and 3) given a test interval, return a segment which intersects that test interval or return nil if there is no such segment. This data structure is used in the new connected components algorithm. It can also be used to improve other existing algorithms for computational geometry problems.

*Submitted to the Department of Electrical Engineering and Computer Science on August 30, 1985 in partial fulfillment of the requirements for the degree of Master of Science. This work was supported by the Defense Advanced Research Projects Agency under contract number N00014-80-C-0622.

**Department of Electrical Engineering and Computer Science, MIT, Room NE43-834, Cambridge, MA 02139, (617) 253-2345.

SPACE-EFFICIENT ALGORITHMS FOR COMPUTATIONAL GEOMETRY

by

CYNTHIA A. PHILLIPS

B.A., Harvard University
(1983)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements for the
Degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1985

Signature of Author_____
Department of Electrical Engineering and Computer Science
August 30, 1985

Certified by_____
Prof. Charles E. Leiserson
Thesis Supervisor

Accepted by_____
Prof. Arthur C. Smith
Chairman, Departmental Graduate Committee

Space-Efficient Algorithms for Computational Geometry

by

Cynthia A. Phillips

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 1985 in partial fulfillment of the
requirement for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

This thesis presents an algorithm for determining the connectivity of a set of N rectangles in the plane, a problem central to avoiding *aliasing* in VLSI design rule checkers. Previous algorithms for this problem either worked slowly with a small amount of primary memory space, or worked quickly but used more space. The algorithm presented here, based upon a technique called *scanning*, operates in $O(N \lg N)$ time in the worst case. This matches the running time of the best known sequential algorithm for this problem. Because we use a machine model that explicitly incorporates secondary memory, the new connected components algorithm avoids unexpected disk thrashing which leads to lower performance. The algorithm uses $O(W)$ primary memory space, where W, the *scan width*, is the maximum number of rectangles to cross any vertical cut. It requires no more than $O(N)$ transfers between primary and secondary memory.

When a vertical line passes through a set of rectangles, those rectangles cut by the line form a set of line segments. The key to development of space–efficient algorithms using a two layer memory model is that appropriate manipulations of these segments alone can solve more complicated problems such as the connected components problem. This thesis introduces *interval trees*, a simple, sparse, data structure for storing a set of k line segments. With this data structure, a variation on a balanced search tree, one can perform each of the following operations in $O(\lg k)$ time in the worst case: 1) insert a new segment, 2) delete a segment, and 3) given a test interval, return a segment which intersects that test interval or return nil if there is no such segment. This data structure is used in the new connected components algorithm. It can also be used to improve other existing algorithms for computational geometry problems.

Thesis Supervisor: Charles E. Leiserson, Associate Professor of Electrical Engineering and Computer Science

Keywords: computational geometry, design rule checking, VLSI, algorithms, rectangles, connected components, scanning, data structures, balanced trees.

Acknowledgements

First of all, a gigantic thanks to my advisor Charles Leiserson. He provided most of the original ideas for the connected components algorithm. He spent many hours working on the presentation style of the first chapter much of which appears in a joint paper. He also is an oracle of information, an objective and interested listener, and perhaps most importantly, a source of encouragement.

Thanks also to Ray Hirschfeld who has helped me through MIT bureaucracy and through the incomprehensible jungle of old TEX. Alex Ishii made many helpful comments on an earlier version of this write up. Thanks to all the other members of the theory group who listened to preliminary versions of this work. Finally, thanks also to my husband Tom for cooking my meals, helping with last minute illustrations, and putting up with me through these last few weeks.

While I was working on this research, I was supported by several research assistantships under Defense Advanced Research Projects Agency contract N00014-80-C-0622.

TABLE OF CONTENTS

Introduction

For a VLSI design to be reliably produced as a working chip, various features on the chip must be separated by minimum distances to ensure the proper operation of transistors and interconnections. The design rule checker program verifies that these and other geometric constraints are satisfied and signals an error if it finds two features that violate the design rules. For a chip composed of millions of rectangles, design rule checking is a time-consuming process which cannot be done entirely within the primary memory of many computers.

This thesis presents an efficient algorithm for finding the connected components of rectangles in the plane using a machine model which incorporates the secondary disk memory where the VLSI design is stored. By running this algorithm simultaneously on each layer of a VLSI chip design, a design rule checker can determine which features of a chip design are electrically equivalent, *i.e.*, are effectively part of the same wire. The determination of electrical equivalence allows the design rule checker to avoid reporting the many *aliasing* errors which occur when two electrically equivalent features are mistaken for electrically distinct features. For example, two wires might be too close together, but if they are actually the same wire, it does not matter.

Many VLSI design systems use rectilinearly oriented rectangles to represent the design features. Two rectangles are electrically equivalent if they are connected by a path of intersecting rectangles. The connected components problem is to label each rectangle in a design such that two rectangles have the same label if and only if they are in the same connected component. The set of rectangles in Figure 1, for instance, has three connected components: $\{A, B, D, E, G\}$, $\{C, F\}$, and $\{H\}$.

The connected components of N rectangles in the plane can be determined in $O(N \lg N)$ time by an algorithm due to Guibas and Saxe [4]. Their algorithm uses the technique of *scanning*, introduced by Shamos and Hoey [8], which assumes that the vertical edges of rectangles are initially sorted by x-coordinate. Scanning algorithms work by sweeping a *scanline* over a set of geometric objects in the plane and then working primarily with the objects crossed by the scanline. In the Guibas-Saxe algorithm, the scanline is a vertical line that sweeps from left to right over the rectangles (Imai and Asano [5] also have an $O(N \lg N)$ connected components algorithm for the primary memory model which is not based on scanning.)

The $O(N \lg N)$ running time that Guibas and Saxe achieve is remarkable in that there may be as many as order N^2 rectangle intersections. Unfortunately, the Guibas-Saxe algorithm is designed to run entirely within primary memory, and it may cause disk thrashing for a large VLSI chip.

In this thesis, we abandon the simple primary memory model, and instead use a machine model which includes a secondary disk memory as well as primary memory. The configuration is shown in Figure 2. We assume that the primary memory is a fast, random-access memory of limited size. The set of rectangles is kept in a file in secondary disk storage. Accesses to the file are presumed to be sequential, either forward or backward. More general random accesses to disk blocks are unnecessary for our algorithm.

This model is used by Szymanski and Van Wyk [10] for a connected components algorithm, a special case of their algorithm for connectivity analysis of more general regions. Their algorithm is more suitable for large rectangle databases because it uses less primary memory than the Guibas-Saxe algorithm and has locality of reference for secondary memory. The amount of primary memory space used by the algorithm is $O(W)$, where W, the *scan width*, is the largest number of rectangles cut by any scanline. In practice, Szymanski and Van Wyk comment, the size of W is about $O(\sqrt{N})$. Unfortunately, their algorithm is based on rectangle intersections, and the running time can be as large as $O(NW)$.

This thesis presents a connected components algorithm that combines and optimizes the Szymanski and Van Wyk and the Guibas and Saxe algorithms. It uses $O(W)$ (primary memory) space and runs in $O(N \lg N)$ time in the worst case.

The algorithm consists of a two-pass scan over the set of rectangles. Most of the work is done in the first, *forward* scan. A *backward* scan is then used to produce the labeling of rectangles such that two rectangles have the same label if and only if they are in the same connected component. The algorithm maintains four data structures of size $O(W)$ during its forward scan.

The first chapter of this thesis presents the connected components algorithm and its analysis. The second chapter describes a data structure and algorithms for the implementation of the *scan set*, one of the data structures used in the forward scan of the connected components algorithm. Some sort of scan set appears in every scanning based algorithm for solving problems with rectangles. In particular, the new data structure, *interval trees*, could be used by Guibas and Saxe to simplify the implementation of their algorithm. The introduction to chapter 2 discusses some the the previous implementations of scan sets.

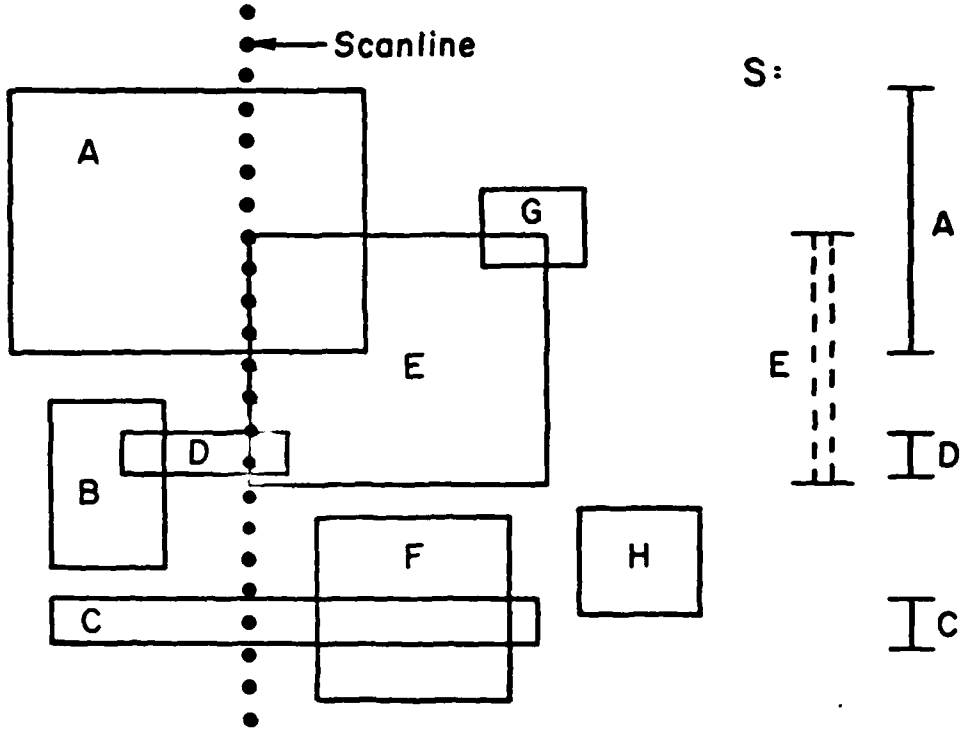


Figure 1: A set of rectangles with connected components $\{A, B, D, E, G\}$, $\{C, F\}$, and $\{H\}$. On the the right is shown a scan set at the time rectangle E enters. Only active rectangles (those crossed by the scanline) have an interval in the scan set. The interval for E will be entered in the scan set S after all processing for its enter event is complete.

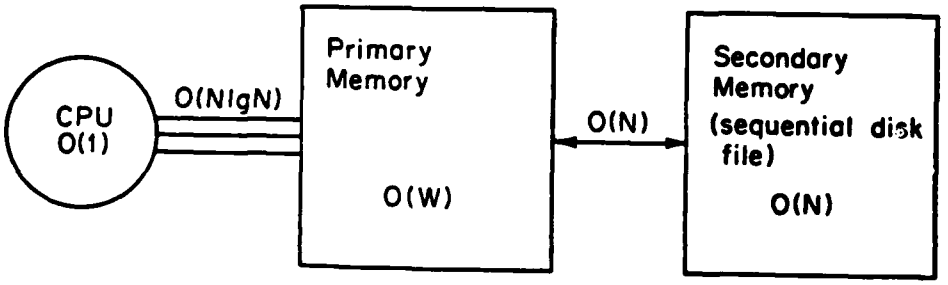


Figure 2: The computer model includes a secondary disk memory as well as primary memory. The connected components algorithm, which assumes the rectangle database is on disk, uses $O(N)$ references to sequential files, $O(W)$ primary memory space, and $O(N \lg N)$ CPU time.

1. The Connected Components Algorithm

This chapter presents the connected components algorithm and its analysis. Sections 1, 2, 3, and 4 describe the four data structures used during the forward scan. Section 5 gives the algorithm, section 6 proves its correctness, section 7 analyses the time and space requirements of the algorithm, and Section 8 offers some remarks.

1.1. Rectangle set

In scanning algorithms an *event* is a geometric phenomenon that causes some computation at the time when it occurs. There are two types of events for a left-to-right scan: a *start event* when the scanline crosses the left boundary of a rectangle (the rectangle becomes *active*, or *enters*) and an *end event* when the scanline crosses the right boundary of a rectangle (the rectangle becomes *inactive*, or *leaves*). Each rectangle has an associated start event and end event.

There are two technical issues to be resolved during scanning. The first is management of active rectangles, and the second is the sorting of events.

The *rectangle set* R is a dynamic set that contains the active rectangles at any point during the scan. The problem is that the association between start and end events must be maintained in small primary memory space. We assume that each rectangle in the disk file has a unique identification number. When a rectangle enters primary memory, it is stored in the set R with the identification number as a key. The rectangle set can be maintained as a balanced search tree, using $O(W)$ space. Each insertion, deletion, or search takes $O(\lg W) = O(\lg N)$ time.

The scanning part of our algorithm assumes the events are sorted by x-coordinate. If not, the events must be sorted. This takes $O(N \lg N)$ time in the worst case, but most computer systems do have a fast disk sort. Much of the time, we can do much better because many VLSI databases already keep rectangles sorted by left edge.

Given a file sorted by left edge alone, we can sort it into start and end events in $O(N \lg N)$ time and $O(W)$ space using an idea due to Szymanski and Van Wyk [10]. The idea is to keep a priority queue, such as a heap [1, p. 147-152], in primary memory. During the operation of the algorithm, the priority queue holds at most $W+1$ rectangles sorted by right endpoint. When a new rectangle is read in, its right endpoint is stored in the priority queue. Then the priority queue is emptied of all rectangles with right endpoint smaller than the left endpoint of the new rectangle. For each of these rectangles, the right endpoint is written out in order as an end event. Then the left endpoint of the new rectangle is written out as a start event. Thus, without loss of generality, we can assume the start and end events are presorted.

There are other, more mundane data management issues to be faced in the course of programming the connected components algorithm described here. Most of these can be resolved using simple pointer associations, but the more complicated will be addressed directly in the sections to come.

1.2. Scan set

We now turn our attention to the data structure that maintains the scanline for the connected components algorithm. At any point during the forward scan, the active

rectangles can be represented as a set of vertical intervals, *i.e.*, an interval in y. For example, Figure 1 shows the intervals of the active rectangles at the time rectangle E enters. The *scan set* S maintains the dynamic set of intervals that represents the active rectangles.

The scan set allows the connected components algorithm to determine rectangle intersections easily. Two rectangles intersect if and only if there is a scanline that crosses both rectangles, and their intervals overlap in the scan set corresponding to the scanline. This technique for determining rectangle intersections is well known and is used in previous scan-based algorithms for determining rectangle intersections or connected components [3], [4], [10].

To be precise, a scan set S supports the following operations:

S-INSERT!(A)
: Add rectangle A to the scan set.

S-DELETE!(A)
: Remove rectangle A from the scan set.

S-FIND(I)
: Returns a rectangle in the scan set S that overlaps interval I in some way, and NIL if no rectangles overlap I.

The number of rectangles stored in S at any given time during a scan is at most the scan width W. We can implement each of the three operations in time $O(\lg W)$ using space $O(W)$ with interval trees. This data structure is described in chapter 2.

1.3. Component set

During the forward scan, the connected components algorithm maintains a *component set* Q that reflects our current knowledge of the connectivity of the active rectangles. Each component is designated by a *color*, which for convenience is represented as an integer.[1]

The rectangle colorings within the component set Q may change with a start event. If a new rectangle connects two previously unconnected components, we merge them within the component set Q by recoloring active rectangles in the smaller of the two.

The component set Q supports the following operations:

COLOR!(A)
: Assigns rectangle A a new (unused) color.

UNCOLOR!(A)
: Dissociates rectangle A from others of its color. If A is the last of its color, the color is destroyed (made available for reuse).

COLOR(A)
: Returns A's color.

REPRESENTATIVE(q)
: Returns any rectangle having color q. If there is no such rectangle, return NIL.

[1] The letter Q is mnemonic for "connected components" and "color." The first letters of the alphabet are reserved for rectangles.

RECOLOR!(q_1, q_2)

Takes all rectangles of color q_1 and color q_2 and makes them all either color q_1 or color q_2. The other color is destroyed.

We implement the component set Q using a vector in which each color is represented as an index in the vector. Each slot in the vector contains a pointer to the first rectangle in a doubly linked list of all rectangles of that color, and the number of rectangles in the list. The pointers to implement the linked lists can be stored with the actual rectangles. Each rectangle also stores the index of its color. If the number field is zero, the color is unused, and we then use the pointer field to implement a free list of the unused colors. An extra variable is needed to store the head of the free list.

All operations except RECOLOR! can be implemented in constant time. If we always merge the color with the smaller number of rectangles into the one with the larger number, then we can do $O(N)$ recolorings in $O(N \lg N)$ time. There are at most W rectangles in the component set Q at any given time so the data structure need only be size $O(W)$.

1.4. Territory set

To achieve an $O(N \lg N)$ worst case running time for the connected components algorithm, we must find a way to maintain the component set Q without looking at every intersection. Figure 3 shows the basic idea. The active rectangles B, C, and D have the same color, say 1. The new rectangle E intersects all three of these rectangles, which tells us that rectangle E should be given the same color as rectangle B, all rectangles with B's color should be merged with rectangles of rectangle C's color, etc. We would get the same result, however, if we just noticed that rectangle E intersects some rectangle(s), all of color 1. That is, instead of asking, "What other rectangles does rectangle E intersect?" we would like to be able to ask, "For what color q in the component set Q does rectangle E intersect at least one rectangle colored q?" We now describe a new data structure called a *territory set* T that will allow us to answer this question using small space and time.

The territory set T is a refinement of the *illuminator* data structure used by Guibas and Saxe in their algorithm for the connected components problem [4]. The territory set is essentially a colored partition $\{t_i\}$ of the scanline. Conceptually, each territory has two fields: its *interval* and its *color*. The interval is a closed interval in y. We implement the color indirectly by associating with each territory a *representative rectangle* which is in the territory, and therefore has the same color as the territory. Each territory t in T obeys the following rules:

1. Each active rectangle is covered by exactly one territory.
2. Each territory covers at least one active rectangle. To ensure that the territory set is never empty, we assume there is a dummy rectangle above all rectangles in the data base that extends the full length of the design.
3. All active rectangles covered by territory t have the same color as t.

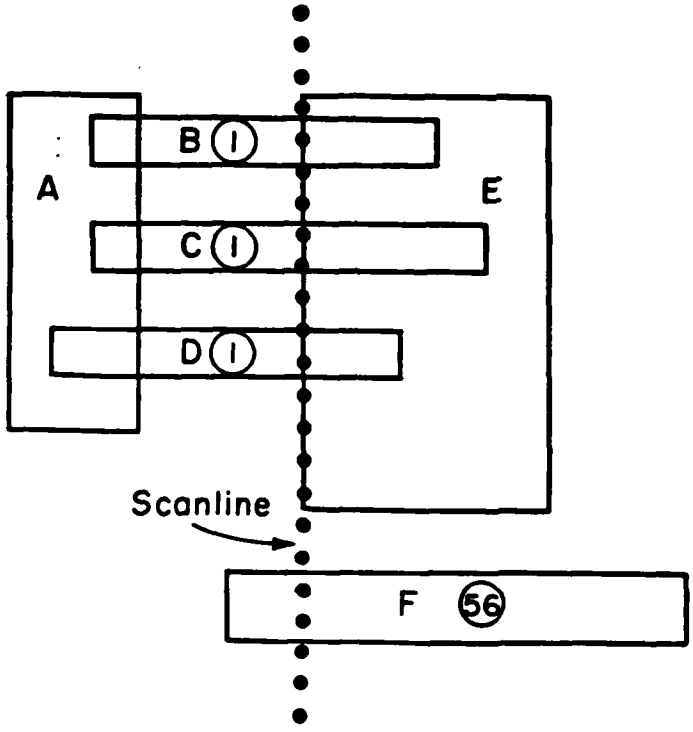


Figure 3: The inefficiencies that can arise from intersection-based connectivity algorithms. Colors of active rectangles are represented as circled numbers. When rectangle E enters we would like to know it should have the same color as each rectangle colored 1 without recognizing this fact three different times via intersection checks.

For example, in Figure 4 no rectangles go across the boundary between territories t_1 and t_2. Each territory covers at least one active rectangle. Each active rectangle's color corresponds to the color of the territory that covers it. Here, rectangles A, E and G and territory t_1 that covers them are colored 17. Rectangles C and F and territory t_2 are colored 42.

The territory set T supports the following operations:

T-INSERT!(t)
: Add territory t to the territory set.

T-DELETE!(t)
: Delete territory t from the territory set.

LOCATE(y)
: Returns the territory that includes the y-coordinate y. If the point y falls on the boundary between two territories, the lower of the two is returned.

NEXT(t)
: Returns the territory immediately above territory t.

COLOR(t)
: Returns the color of territory t. This operation involves getting t's representative rectangle and getting the color from the rectangle.

The territory set T can be implemented as a standard height-balanced tree using $O(W)$ space. The operations T-INSERT!, T-DELETE!, LOCATE, and NEXT can each be implemented in $O(\lg W) = O(\lg N)$ time. As a simple optimization, the territories can be linked in order, which allows NEXT to run in constant time.

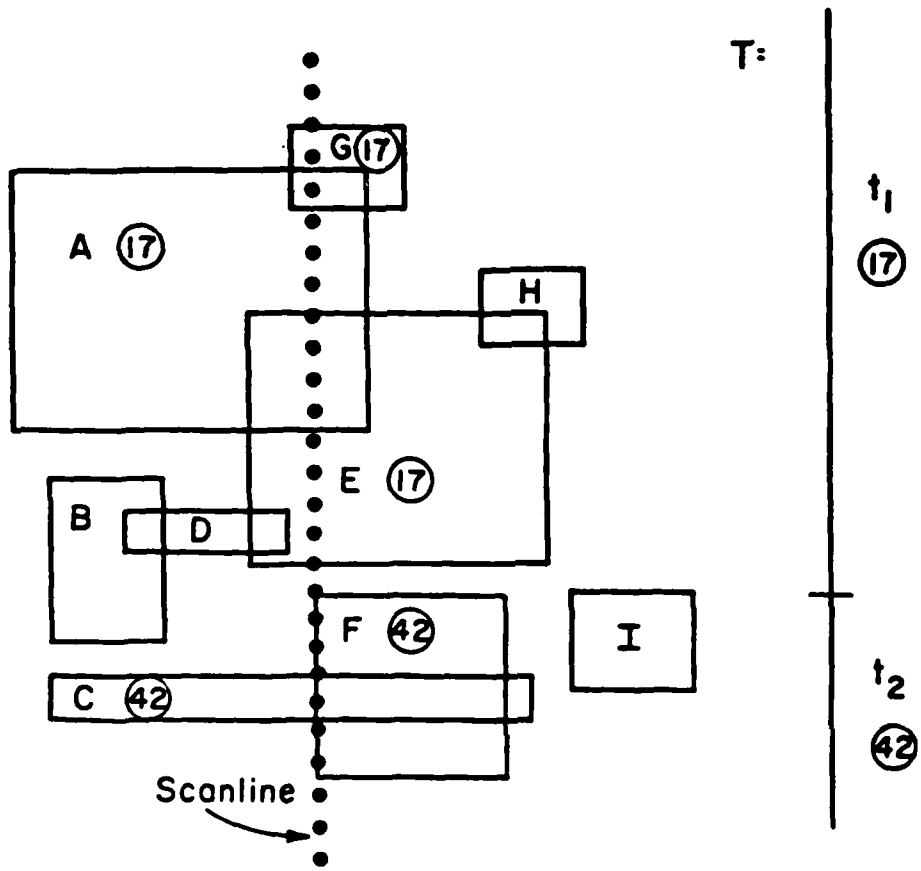


Figure 4: The territory set (right) for a collection of rectangles (left) is essentially a colored partition of the scanline. Colors of active rectangles and territories are represented as circled numbers.

1.5. The Algorithm

The connected components algorithm operates in two phases. The first phase is a forward scan over the rectangles during which connectivity information is prepared that is written out to an intermediate sequential file on disk. The second phase is a backward scan over the intermediate file during which component labels are assigned to each rectangle.

1.5.1. The forward scan

The data structures used by the forward scan contain only those rectangles that are active, which ensures that the $O(W)$ space bound is met, but which also leads to problems maintaining connectivity across the entire database. When we see an end event for a rectangle A signaling that A is to become inactive, we are not prepared to give A a final label, yet we must purge A from our internal data structures. For example, at the time rectangles A and C in Figure 5 become inactive, there is no way to guess that they are in the same connected component. Were we to give them final labels now, we would incorrectly give them distinct labels.

Since we cannot give each rectangle A a final label in the forward scan, we give it a *friend*. Rectangle A's friend is another rectangle which (1) is active at the time rectangle A leaves, and (2) is known to be in the same connected component as rectangle A. If there is no such rectangle at the time rectangle A leaves, then its friend is NIL. Figure 5 shows a possible assignment of friends.

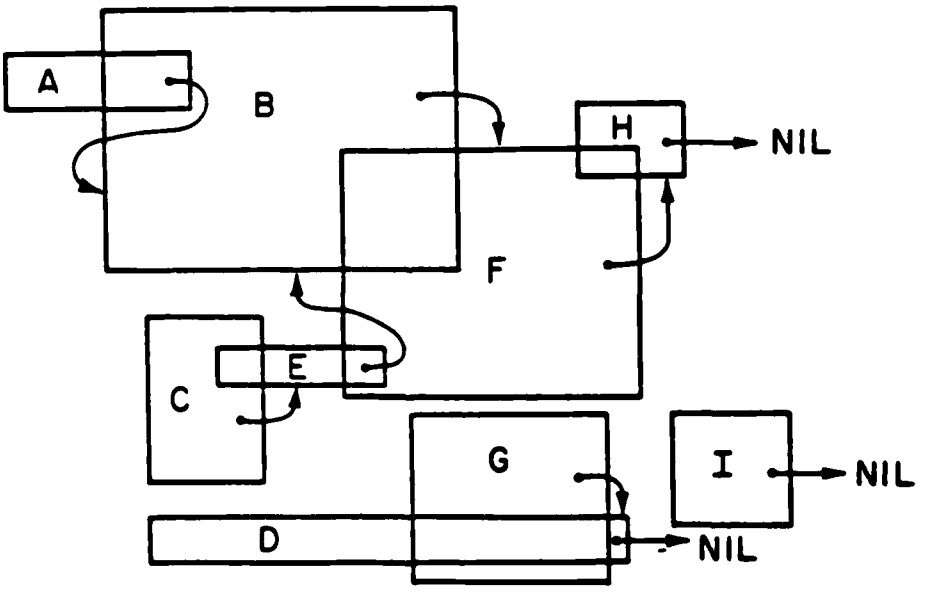


Figure 5: Each rectangle A picks a friend that is active at the time A leaves and is known to be in the same connected component.

At the end of the forward pass, each connected component is linked together by a tree of friend arrows. From this friend information, the back pass can construct final component labels. The idea is that each friend arrow points from left to right if the source and destination rectangles are sorted by right edge, or equivalently, by time of exit. Thus, a component label assigned to the root of the tree will propagate right to left through the tree during the back scan.

The start event

Processing a start event for rectangle A during the forward scan involves four steps: setting up, handling top and bottom boundary conditions, recoloring affected rectangles, and cleaning up.

Set up. Figure 6 shows the important y-coordinates and intervals for the general case. The bottom and top coordinates of A's interval are designated y_{bot} and y_{top}. The endpoints of the k territories in the territory set T that A overlaps are $y_0, y_1, \ldots, y_k$. The k territories are gathered into a list L by first using LOCATE to find the territory that includes y_{bot}, and then using NEXT to gather the remaining territories that overlap A's interval $[y_{bot}, y_{top}]$. All the territories in L are then removed from T, which leaves a gap in T from y_0 to y_k. This gap will be repaired in subsequent steps.

Intuitively, the colors of the territories in list L represent our first guess at which colors must be merged due to the entrance of rectangle A. Since each territory contains at least one active rectangle, the territories in the middle of the list will necessarily contain a rectangle that intersects A.

Handle boundary conditions. Rectangle A extends only partially into the top and bottom territories, so we must explicitly reference the scan set S to determine whether there are active rectangles in these two territories that intersect A. We describe only the handling of the top boundary condition since the bottom boundary condition is symmetric. Also, for simplicity, we shall consider the special case $k = 1$ (Figure 7) after we deal with the general case $k \geq 2$ (Figure 6).

Handling the top boundary condition for $k \geq 2$ involves determining whether the top territory should be kept in list L. The first case is when there is some active rectangle B that intersects the interval $[y_{k-1}, y_{top}]$. The interval of the rectangle B falls entirely within the top territory, so it follows that A, B, and every other active rectangle covered by the top territory must have the same color by the time we finish processing the entrance of A. Therefore, we leave the top territory in the list L, and nothing is to be done.

Otherwise, no active rectangle intersects A in the top territory, and the top territory is removed from L. Since k is at least 2, there must be an active rectangle in the interval $[y_{top}, y_k]$ because the top territory must contain at least one rectangle, and the interval $[y_{k-1}, y_{top}]$ contains none. Therefore, we can return the top territory to the territory set with the shortened interval $[y_{top}, y_k]$ without violating any of the properties a territory must have. In other words, chopping off empty space does not hurt.

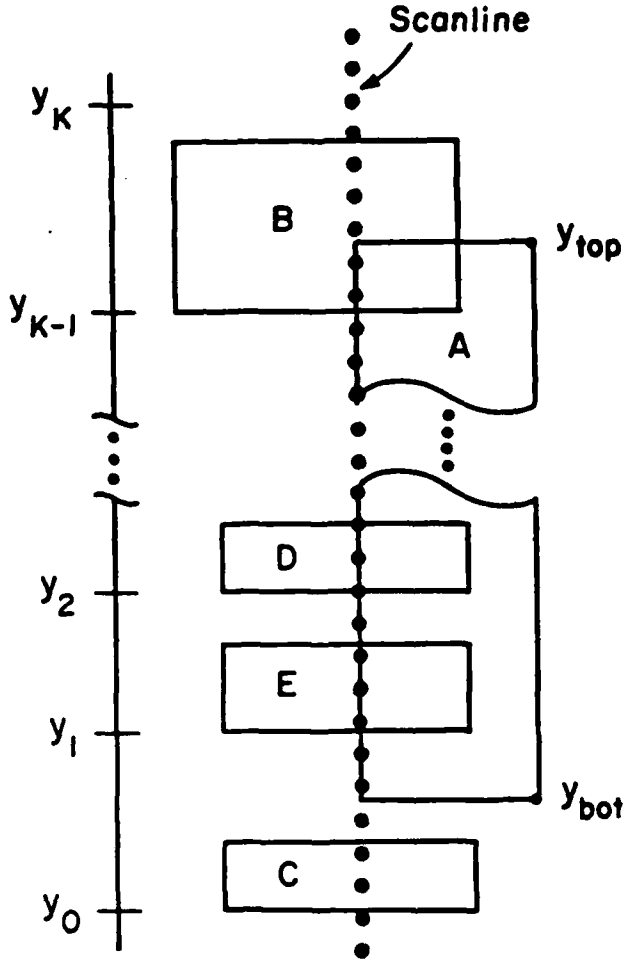


Figure 6: The territory set is shown on the left and the rectangles on the right for the case when $k > 1$. The colors of territories $t_1, t_2, \ldots, t_k$ are a first guess at the colors to merge because of rectangle A's entrance.

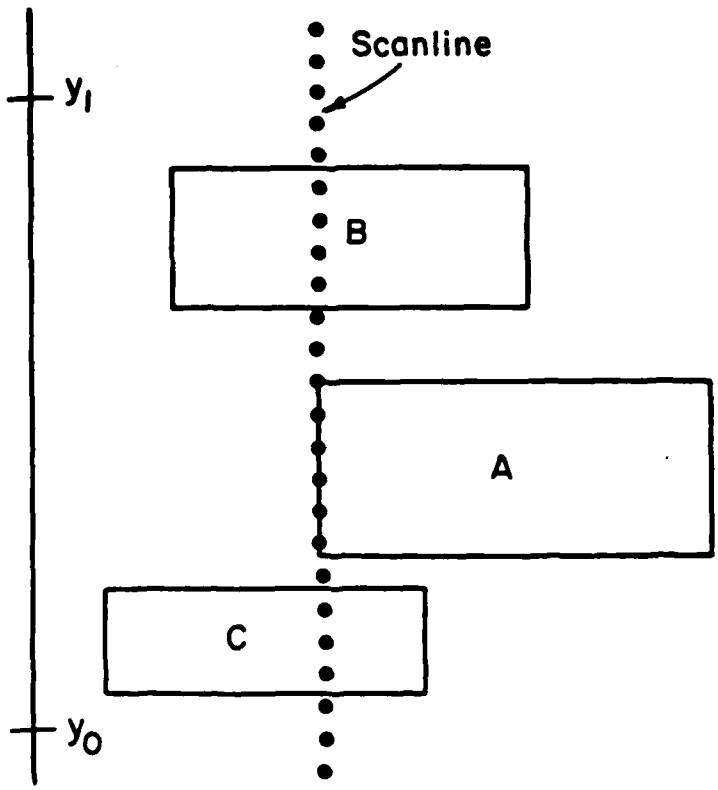


Figure 7: The territory set for the case when $k = 1$. Since rectangle A falls in only one territory, the intervals above and below rectangle A must be checked explicitly.

We now discuss the processing of the top boundary condition for the special case when $k = 1$ (Figure 7), since once again, the bottom boundary condition is symmetric. If rectangle A intersects some active rectangle, then we're done. If rectangle A does not intersect any active rectangle, it is possible that the rectangle that justified the existence of the single territory in list L is below rectangle A, instead of above. In this case, we must explicitly query the scan set S with the interval $[y_{top}, y_1]$ to determine whether there is an active rectangle to justify putting a territory over the interval. If there is an active rectangle, we must enter a new territory into T with the shortened interval $[y_{top}, y_1]$ using the color of the old territory.

Recolor. Now, the colors of the territories in list L are exactly the colors that must be merged because of rectangle A's entrance. We first color rectangle A with a new color. We then merge A's color with the color of each territory in L. The colors are automatically garbage collected by the component set Q. Because of our pointer implementation of territory colors, no territory will ever be colored with a garbage-collected color.

Clean up. We finish the servicing of rectangle A's entrance by repairing the territory set T and making A active. The gap left after handling boundary conditions becomes the interval of a new territory with the color of rectangle A. Rectangle A is inserted in the rectangle set R and the scan set S. Since the left side of a rectangle indicates an end event in the back scan, enter an end event for rectangle A in the intermediate file that will serve as input to the back scan.

The end event

Servicing an end event for rectangle A requires us first to find the associated rectangle object for A in the rectangle set R. Then, we must output a start event for A in the back pass and fix up the internal data structures. We accomplish this processing in three steps: making rectangle A inactive, associating A with a friend, and fixing the territory set T.

Make A inactive. Let q be rectangle A's color before processing the end event. Uncolor rectangle A, and remove it from the scan set S and the rectangle set R.

Find a friend. Query the component set Q for a representative of color q. Associate this representative rectangle (possibly NIL) with rectangle A so that A can now tell its friend when asked. Write out this information as a start event for rectangle A for use in the back scan. We shall say that rectangles that recieve NIL as a friend have no friend or are friendless.

Fix the territory set. Pick any point on rectangle A's interval, and use LOCATE to find the one territory t that covers rectangle A. Find a rectangle B in the scan set S that intersects t's interval to see if there is some active rectangle to justify t's existence. (Recall that a territory must cover at least one active rectangle.) If no rectangle exists, then A is the last active rectangle in t's interval, and territory t can be eliminated by extending the interval of the next territory above t to include t's interval.

If the existence of territory t is justified by some active rectangle B, and A is serving as the representative for territory t, then make rectangle B the representative of territory t.

1.5.2. The back scan

The second phase, the back scan, passes backwards through the intermediate file of rectangle-friend information created in the forward scan, producing a final file of rectangle-label pairs which will be sorted by left edge. During this right-to-left scan, each rectangle receives its final labeling from its friend. The back scan uses only one data structure, the rectangle set R. It also requires a counter initialized to 0.

During the back scan, the rectangle set R holds all active rectangles, and each active rectangle knows its final component label. Labels are assigned sequentially during the back scan, and the counter holds the value of the next label to be assigned.

The start event

The first step in servicing a start event for rectangle A is to assign a final label to A. If A has no friend, it is the rightmost rectangle in its component, and so a new label must be assigned from the counter. Store this label into A and increment the counter.

Otherwise, find rectangle A's friend in the rectangle set R, and give A the same label as its friend. Rectangle A's friend must be active since A and its friend were simultaneously active in the forward scan. Rectangle A left first in the forward scan so it must enter after its friend in the back scan. Finally, add rectangle A to the rectangle set R.

The end event

Processing an end event for a rectangle A consists of simply removing A from the rectangle set R and writing out rectangle A with its label to a final file. No rectangle that subsequently enters has A as a friend because the two rectangles are not simultaneously active. Thus, no other rectangle will need to get a label from A, and hence, it is safe to remove A. The final file is sorted by left edge from right to left. Reversing the file leaves it file sorted left to right by left edge as was the original input file.

1.6. Proof of correctness

This section shows that two rectangles get the same label if and only if they are in the same connected component.

($\Rightarrow$) We first show that if two rectangles are given the same label, then they are in the same connected component. We prove this by induction on the number of rectangles given the same label. Suppose rectangle A is the first rectangle given label l. Then at the time we process rectangle A's start event during the back scan, rectangle A is friendless and the value of the counter is l. If rectangle A had a friend, it would be given the same label as its friend contradicting our assumption that rectangle A was the first to receive its label. The counter is incremented after rectangle A is given the label l so no friendless rectangles to enter after A will get the label l. By the same argument, no friendless rectangle to enter before rectangle A could have been given the label l.

Assume that at some point in the backscan, k rectangles have been given label l and all k are in the same connected component. Some $j \leq k$ of these rectangles

are active (i. e. are in the rectangle set R). Now the start event for some rectangle B causes B to get label l. For this to happen, rectangle B must have a friend rectangle C which is one of the j active rectangles with label l. Since rectangle C is rectangle B's friend, both rectangles B and C must have had the same color in the component set Q at the time rectangle B left in the forward scan.

To finish the argument, we must show that two rectangles simultaneously having the same color in the component set Q must be in the same connected component. If this is true, then rectangles B and C are in the same connected component and therefore by transitivity rectangle B is in the same connected component as the other k rectangles given label l.

We show that two rectangles sharing a color in the component set Q must be in the same connected component by induction on the number of rectangles with that color. A new color is introduced into the component set Q only when a COLOR! operation is performed upon a rectangle A during its start event. Hence each color begins with only one member rectangle. Other rectangles join a color only through the MERGE! operations performed during the processing of the start event for a rectangle.

Assume that before processing the start event for a rectangle A, all rectangles with the same color in the component set Q are in the same connected component. After handling the boundary conditions, there are $m \geq 0$ territories in the list L. These territories have $n \leq m$ distinct colors $q_1, q_2, \ldots, q_n$ which are all merged into one final color q. The colors $q_1, q_2, \ldots q_n$ are exactly those colors for which at least one member rectangle intersects rectangle A. Each pair of rectangles in the final color q is connected by a path of intersecting rectangles. If they shared a color q_i in the component set Q before rectangle A entered, then by assumption there is a path connecting them that includes only rectangles originally colored q_i. Otherwise, there is a path between the two rectangles that includes rectangle A. Therefore all rectangles now colored q are in the same connected component.

($\Leftarrow$) We now prove that if two rectangles are in the same connected component, then they get the same label. It suffices to show that if two rectangles intersect they get the same label because then all rectangles in the same connected component get the same label by transitivity. The proof has two parts. First, we argue that if two rectangles intersect, then during the forward scan they have the same color in the component set Q while they are both active. Then we show that if two rectangles are simultaneously active and have the same color in the component set Q then they get the same label.

If two rectangles A and B intersect, they have the same color in the component set Q while they are both active. Assume without loss of generality that rectangle B enters after rectangle A. Let t be the territory in the territory set T that covers rectangle A at the time rectangle B enters. Since rectangles A and B intersect, territory t must at least partially cover rectangle B so it will be gathered into the list L in the first step of the processing of the start event for rectangle B. The presence of the active rectangle A intersecting rectangle B guarantees that after the boundary condition checks, territory t will still be in the list L. Therefore after the merging, rectangles A and B will be the same color in the component set Q. From that point on they will always move together in any recolorings, so they will always have the same color until one of them leaves.

If two rectangles are in the same color in the component set Q while they are active, then they will get the same final label. Suppose a rectangle A is about to leave. Consider the set of rectangles that have the same color as rectangle A. Rectangle A chooses one as a friend (shown by an arrow in figure 8). Later, other rectangles may join this set through merges. As each rectangle in the set leaves, it chooses a friend from among those left in the set. Eventually an exiting rectangle finds itself alone, and it exits without a friend.

Figure 8 shows one such set of rectangles taken from the example in figure 5. During the forward scan each of these rectangles simultaneously shares a color in the component set Q with at least one other rectangle in the set. For example rectangles A and B share a component immediately after rectangle B enters and rectangles $B, E,$ and F share a component immediately after rectangle F enters.

We can view the illustration in figure 8 as an acyclic graph with the rectangles as vertices and the friend relation arrows as directed edges. Each vertex has outdegree one except for a single sink, the friendless rectangle H. If we start at any vertex in the graph and follow the edges, we always end up at the sink. We know from our previous argument that a rectangle gets the same label as its friend. That friend in turn gets the same label as *its* friend, ... (down the friend links) ..., who gets the same label as the sink H. By transitivity any two rectangles that are in the same component of the component set Q while active will get the same final label.

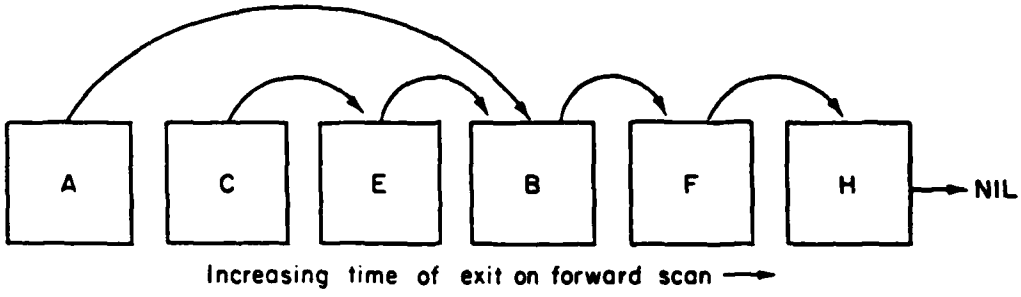


Figure 8: An example of a component taken from figure 5. The arrows represent the friend relation. Each rectangle shares a component color with at least one other rectangle in this set during the forward scan. During the back scan, these rectangles enter from right to left. Rectangle H receives a new label, and all the other rectangles receive their labels indirectly from rectangle H.

1.7. Analysis

This section shows that the worst-case running time of the connected components algorithm is $O(N \lg N)$, the amount of primary memory required is $O(W)$, and the number of transfers between primary and secondary memory is $O(N)$. We have already seen that each data structure requires only $O(W)$ primary memory space, and it can be verified that the number of disk transfers is $O(N)$. Thus, we must demonstrate that the running time of the algorithm is $O(N \lg N)$.

The rectangle set R and the scan set S each contribute only $O(N \lg N)$ to the overall time. The rectangle set R is used in both the forward scan and the back scan. It contributes only $O(N \lg N)$ to the time in each phase since it performs at most two operations, each requiring $O(\lg N)$ time, on each of the $O(N)$ start and end events. The scan set S performs one insertion or deletion and at most four S-FIND operations for each start or end event.

Operations on the territory set T contribute $O(N \lg N)$ time as well. During the servicing of an end event, the territory set T performs at most one LOCATE, two T-DELETE!'s, and one T-INSERT!, if we regard the modification of a territory interval as a deletion followed by an insertion. For a start event, only one LOCATE and at most three T-INSERT!'s are performed. The number of calls to NEXT, T-DELETE!, and COLOR directly depends on the size of the list L, however.

We shall show that each operation is performed at most $O(N)$ times. The operations that are performed a constant number of times for a given event are executed $O(N)$ times overall. The other operations are called once for each time a territory appears in a list L during a start event. Thus, showing that the sum total of the sizes of L throughout the entire forward pass is $O(N)$ will produce our desired bound. The total number of insertions into T is at most $4N$, which therefore bounds the total number of deletions. Moreover, each of these territories can participate in a list L only once since it is deleted from T at that time and replaced by the consolidated territory or a new boundary territory. Hence, the sum total of the lengths of L is $O(N)$, which also bounds the number of times any operation is performed. Since each operation costs $O(\lg N)$ time, the total work performed on the territory set is $O(N \lg N)$.

It remains to analyze the component set Q. Each start event causes one COLOR! operation, and each end event causes one UNCOLOR! and REPRESENTATIVE operation. Using the same arguments as above for the territory set, at most $O(N)$ RECOLOR! and COLOR operations are performed throughout the whole forward scan. Thus, its contribution to the overall running time of the connected components algorithm is also $O(N \lg N)$.

1.8. Remarks

This section presents the important extension of the connected components algorithm to multiple layers. We also discuss some alternative implementations of the data structures which may be better suited to a practical implementation.

The connected components problem of rectangles in the plane presented in this paper is a simplification of the problem faced in computer-aided design of VLSI.

Computing the electrically equivalent rectangles in multiple planar layers of a VLSI design is not much more difficult than the one-layer problem discussed in this paper, even though contact cuts can allow components to snake up and down among layers.

To find the connected components of rectangles on multiple layers, we simply run a copy of the basic, one-layer, connected components algorithm on each layer. In the forward scan each layer is given its own scan set, rectangle set, and territory set. The component set, however, is global to the entire computation. Each contact is represented explicitly on the layers it intersects. In the back scan, both the counter and the rectangle set are global. No further changes are necessary.

Some of the data structures necessary for the connected components algorithm can be implemented more practically than with the asymptotically efficient height-balanced trees presented in the body of the paper. The rectangle set R, for example, can be implemented by hashing on the rectangle identification number, which would lead to good average case behavior. At the cost of a bit more complication, the component set Q can be implemented with a union-find structure that allows $O(N)$ merges in almost linear time [11].

The scan set S and the territory set T can be implemented by using bins, as has been done for other VLSI algorithms [2]. Each bin represents a fixed portion of the scanline and contains a pointer to the list of objects that overlap that bin. A desirable bin size can be chosen based on statistical information about the VLSI design. The worst-case running time of the algorithm may be diminished, however, because long, tall rectangles will be split across many bins. The difference between this approach and an intersection-based approach, such as [10] may be negligible.

2. Interval Trees

The scan set data structure is central to any algorithm that uses scanning. In particular, an algorithm which passes a scanline over a set of rectangles requires a data structure to manipulate the line segments that the scanline induces in that set of rectangles. The INSERT!, DELETE!, and FIND operations that the algorithm of chapter 1 requires are a subset of the operations required by other scan-based algorithms for problems involving rectangles, for example [3, 4].

Other data structures have been developed to handle INSERT!, DELETE!, and the more complicated operation of ennumerating all segments intersecting a given test interval. Each of these data structures has its shortcomings. The segment trees of Bentley and Wood [3] require $O(n \lg n)$ space for n segments. McCreight's priority search trees [6] require only $O(n)$ space, but they are quite complicated. Priority search trees are built upon height balanced trees. Unfortunately, updates after rotations require $O(\lg n)$ time, so the underlying balanced tree structure is limited to those which have a constant number of rotations on each insertion/deletion.

The three operations we want—insert a segment, delete a segment, and find any segment that overlaps a test interval—do not require the heavy artillery of priority search trees. A simple modification applied to *any* height balanced tree scheme will suffice. The new scheme, *interval trees*, requires only $O(n)$ space and it performs each of the three operations in $O(\lg n)$ time.

Section 1 introduces interval trees. Section 2 describes the insert and delete operations. Section 3 gives the algorithm for the find operation and argues that it is correct. Section 4 offers some conclusions.

2.1. The structure of Interval Trees

Interval trees represent a set of line segments, all intervals along the same line. Each segment, s, is represented as an ordered pair (s_1, s_2), $s_1 < s_2$, where s_1 is the minimum point of the line segment and s_2 is the maximum. It is assumed that the minimum points of all segments are distinct. It they are not, break ties with the maximum points or with an ID attached to each segment. All that really matters is that the intervals are distinguishable.

To implement an interval tree, start with any balanced search tree scheme: AVL trees, 2-3 trees, etc. Set up the search tree as it would normally be set up using the minimum point of each segment as the search key. The segments may be stored in internal nodes as in AVL trees or stored strictly at the leaves as in 2-3 trees. Now add to each internal node a *range interval* corresponding to the minimum and maximum points covered by any segment in the subtree rooted at that node. The minimum point of the range interval for an internal node always comes from its leftmost son if the tree is based on any standard search tree. However, the maximum point comes from anywhere in the subtree.

Figure 9 shows one possible implementation of an interval tree for a set of segments. It uses a 2-3 tree strategy where all the segments are stored in the leaves. Figure 10 shows another underlying scheme more like an AVL tree where segments are

stored in internal nodes as well as the leaves. Here the range interval for an internal node takes the minimum and maximum points covered by segments in any part of the subtree including the node itself. For example, in the left son of the root, the maximum point comes from segment c stored in the node itself.

Some points in the range interval for a node may not be covered by any segment stored in the subtree rooted at that node. These uncovered intervals are called *gaps* in the range interval. For example, the node in figure 9 with three children has two gaps in its range interval: h_2 to i_1 and i_2 to j_1.

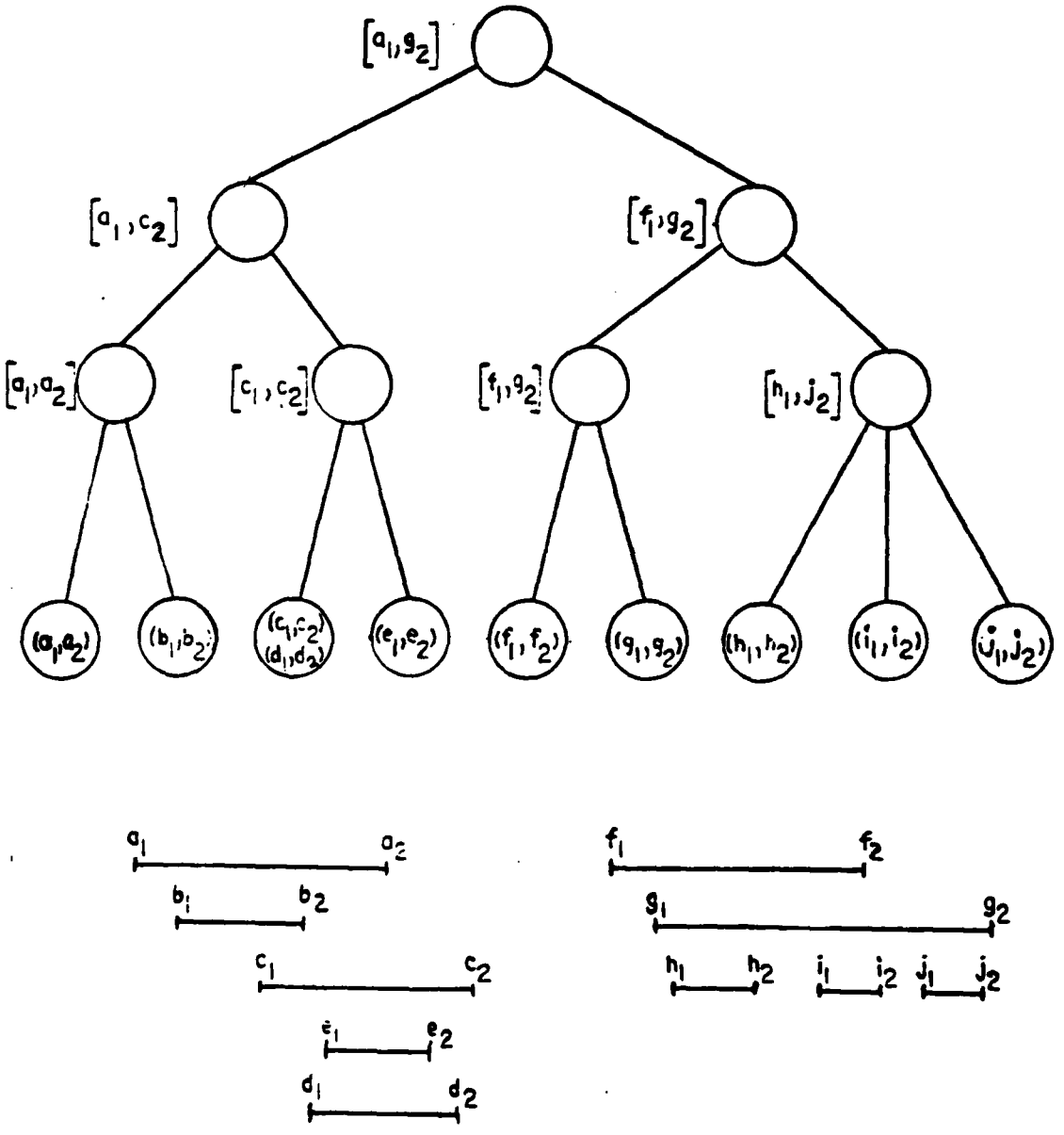


Figure 9: An interval tree with an underlying 2-3 tree representation for a set of segments. Intervals in brackets are range intervals for internal nodes.

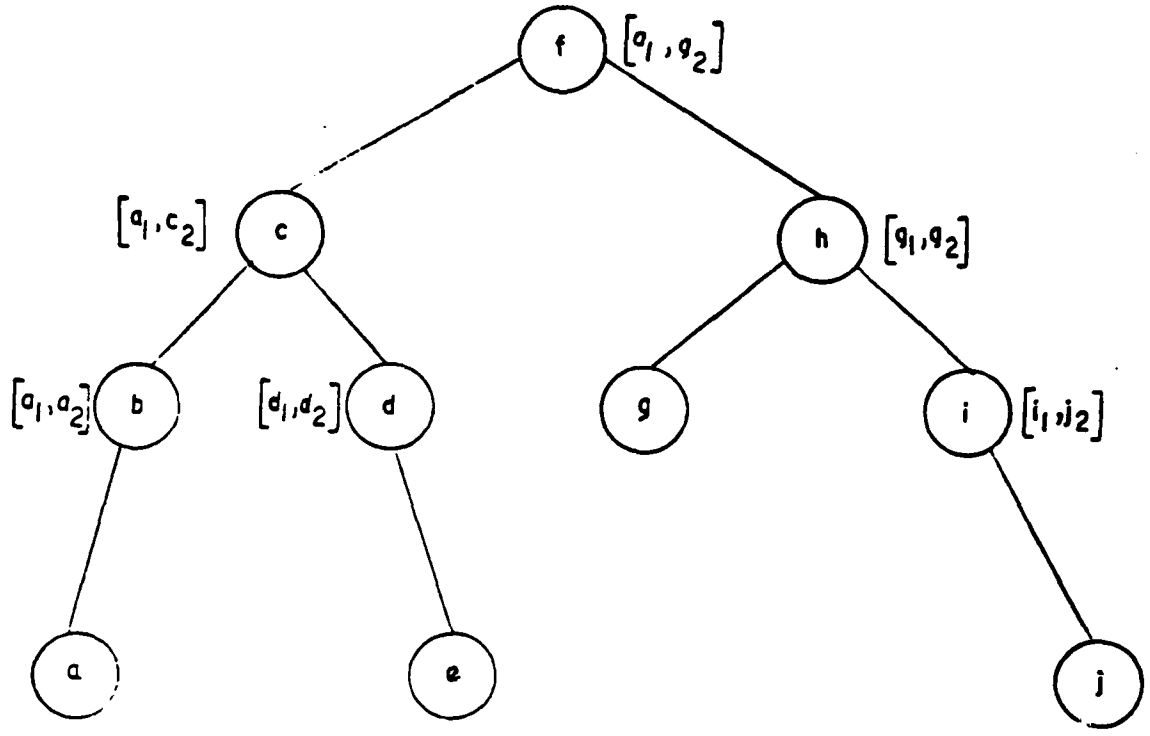


Figure 10: An interval tree containing the segments from figure 9. The underlying balanced search tree here is an AVL-like tree which uses rotations to maintain balance. The letters within nodes indicate segments. For example, g stands for the segment (g_1, g_2). Bracketed intervals are the range intervals for the internal nodes.

2.2. Insertion and Deletion of Segments

This section describes how to insert and delete segments from an interval tree. A segment can be inserted/deleted and the range intervals updated in time $O(\lg n)$ in an interval tree with n segments. We describe the operations in some detail for interval trees implemented with underlying tree structures similar to 2–3 trees and AVL trees.

Insertion and deletion of a segment is done in accordance with the underlying balanced tree scheme. In AVL trees and others like it, balance is maintained through rotations. In B tree schemes such as 2–3 trees, it is maintained by node splitting, sharing keys among siblings, etc. In either case, maintenance of the range intervals is quite easy. This is because once the range intervals of a node's children are established, calculation of its range interval takes a constant amount of time.

Insertion into 2–3 trees involves two steps. The first stage is a search to find the leaf in which the new segment belongs. The segment is then inserted into that leaf. In the second stage, this leaf splits into two leaves if the addition of the new segment made the leaf too full. The splitting of the leaf may make its parent overfull. If so, the parent splits, and so on.

To update range intervals during the first stage, fix effected intervals on the way down the tree during the search for the appropriate leaf. That is, each node passed through in the search will be an ancestor for the new segment, so if the new segment has a minimum point lower than the minimum point of a range interval or a maximum greater than its maximum, adjust the range interval. On the second stage, adjust the interval of each node involved in splitting. The splitting goes from the bottom of the tree up, so each node's children are stable by the time it splits. Therefore calculation of a new range interval requires a constant amount of time for those nodes.

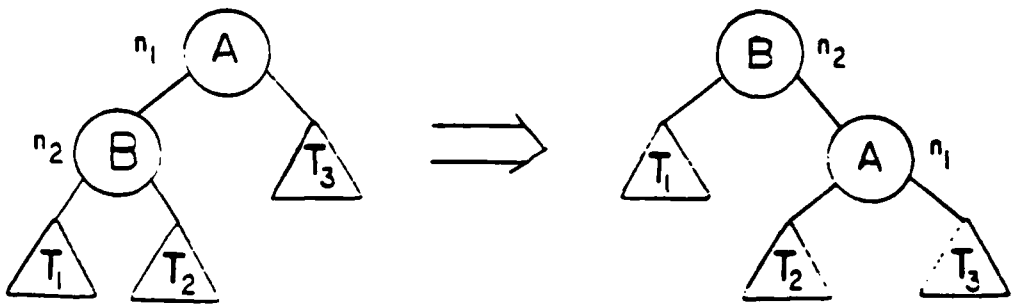


Figure 11: An example of a typical rotation. Circles represent nodes of a tree. Triangles represent tress of 0 or more nodes. This rotation is performed if both of the trees T_2 and T_3 have height h and tree T_1 has height $h + 1$. McCreight uses this example when explaining updates to priority search trees after rotations. Priority search trees require $O(\lg n)$ time for each rotation which restricts the number of balanced tree schemes suitable as underlying structures. Interval trees require only a constant amount of time per rotation.

Deletion is much the same except that instead of splitting, there is coalescing. THe same reasoning applies here: adjust range intervals on the way down while searching for the segment to delete, and adjust each node involved in coalescing.

Trees that use rotations to adjust balance are even easier. As with those interval trees with underlying 2-3 trees, adjust range intervals on the way down the tree when making an insertion/deletion. If the tree is unbalanced, make the necessary rotations. Figure 11 shows a typical rotation. The circles represent nodes with segments stored in them. The triangles represent subtrees of 0 or more such nodes. These rotations cause problems with priority search trees because the subtrees T_1, T_2, and T_3 must change when updating information stored in the nodes n_1 and n_2. With normal search trees and interval trees this is not the case. The range intervals in trees T_1, T_2, and T_3 remain unchanged. Give node n_2 the old range interval from node n_1, and calculate a new range interval for node n_1 using segment A and the range intervals of trees T_2 and T_3. Since rotations take constant time, we can do up to $O(\lg n)$ of them on each insertion/deletion without effecting the assymptotic time requirements for these operations.

2.3. The FIND operation

This section gives the algorithm for the FIND operation required by the scan set of the algorithm presented in chapter 1. It also contains an argument that this algorithm is correct.

Given a test interval $t = (t_1, t_2)$ and an interval tree, FIND(t_1, t_2) returns a segment in the interval tree which *covers* (overlaps) the interval t or it returns nil if there is no such segment. By appropriately defining "covers" we can implement open or closed endpoints in any combination for both the segments and the test interval, (eg. open test interval with half open segments or closed test interval with open segments). We can even vary within the segments in the interval tree if we want to make the processing a bit more complicated and allow two extra bits of information per segment. The difference between open and closed endpoints is merely the difference between a test with a strict "less than" and a test with "less than or equal to".

Given the root of an interval tree, r, and a test interval t, to FIND a covering segment (if any) proceed as follows: If there are any segments stored in node r and one of them covers t, return that segment and halt. Otherwise, look at the range interval of node r's leftmost child. If this range interval covers test interval t, then recursively FIND a covering segment in that child. Otherwise, check the range interval of node r's next leftmost child, and recurse to that node if its range interval covers the test interval t, and so on. If node r contains no covering segment and none of its children have range intervals that cover test interval t, then return nil.

To see why this algorithm works, let us look at a binary tree. The argument for n-ary trees is similar. If we find a covering segment in the node, then we succeed. If there is no covering segment in the node and none of the children have range intervals touching the test interval, then there is no hope; the tree does not contain a covering segment, so we should return nil. The only subtle point is that if the range interval of the left child covers the test interval but a recursive search in the left child fails to

yield a covering segment, then a search of the right child must fail as well. This allows us to follow exactly one path through the interval tree during a FIND operation so the operation costs $O(\lg n)$ time in the worst case in an interval tree with n segments.

Figure 12 shows why the right son will be of no help if the left son's range interval covers the test interval but the recursive search fails. Suppose segments a and b are in the left subtree of a node. During the FIND operation we see the range interval for the left subtree covers test interval t, but the search ultimately fails. For this to happen, interval t must fall into a gap of the left subtree range interval. The only way a segment in the right subtree could cover interval t would be if there were some segment c that reached into the gap. However, that would mean $c_1 < b_1$ which violates the structure of the search tree. If $c_1 < b_1$ then segment c must be in the left subtree if segment b is.

An interesting corollary to this argument is that if the range interval (l_1, l_2) of the left son overlaps the range interval (r_1, r_2) of the right son, then there is some segment in the left subtree that extends through the entire overlap region. The overlap region is either (r_1, l_2) or (r_1, r_2) if $r_2 < l_2$. The segment (p, l_2) which registers the maximum point in the range interval for the left subtree must have a minimum point $p < r_1$ so this segment covers the entire overlap region. Consequently, if during a FIND operation two children of a node have range intervals overlapping the test interval, then we know there is a covering segment and that it is in the leftmost child. If we are not interested in a specific segment, but only care to know if there exists a covering segment or not, we could stop at this point with an affirmative answer. For example, this is all that is necessary in the Guibas and Saxe connected components algorithm.

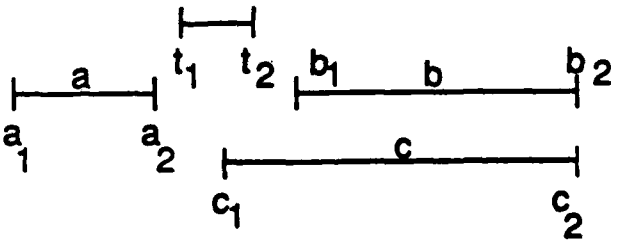


Figure 12: If a and b are segments in a subtree rooted at some node n, then the subtree rooted at node n's right sibling cannot contain a segment c reaching into the gap between segments a and b. That would violate the strict ordering of the search tree based on minimum segment endpoints.

2.4. Remarks

This section wraps up the discussion of interval trees by applying them to the scan set used in the connected components algorithm of chapter 1 and by looking at the problem of ennumerating all segments covered by a test interval.

The scan set of chapter 1 is supposed to contain rectangles, not segments. The problem is easily remedied by using pointers to rectangles whose y interval will serve as the segments. Our insistence on distinguishability is motivated by the scan set. Removing either one of two identical segments from an interval tree would not appear to matter. However, removing the wrong rectangle from the scan set could cause many problems with the connected components algorithm.

The interval tree is sparser than Bentley and Wood's segment trees and simpler than McCreight's priority search trees. Any existing code for a balanced search tree can be modified to implement interval trees in a very short time. Interval trees have only the linear ordering on the lower end of the interval and the very easily maintained range intervals. They do not maintain any heap–type information on maximum points which is what causes all the headaches with priority search trees.

We are taking advantage of the special case operation FIND. In fact, interval trees will not perform as well as segment trees or priority search trees for general ennumeration. All efforts to modify interval trees to perform ennumeration have lead back to variations on priority search trees.

Directions for Further Research

As mentioned in the introduction, the Szmanski–Van Wyk algorithm for connected components assumes the same machine model as the algorithm presented in this thesis. Their algorithm, however, is more general. It is designed for connectivity analysis, computation of union, intersection, etc. of general polygons. Their algorithm uses a two–pass scan assuming an edge file on disk. It runs in $O(NW)$ time where N is the number of edges in the file, and uses $O(W)$ space where W is the maximum number of edges to cut any vertical scanline.

Neivergelt and Preparata have an algorithm for such geometric operations on arbitrary polygons as well [7]. Their algorithm is designed to run entirely within primary memory, but its assymptotic running time is better than that of the Szymanski–Van Wyk algorithm. Given n points in the plane and some connecting line segments with s segment intersections, the Neivergelt–Preparata algorithm runs in time $O((n+s)\lg n)$. In the special case where the points and lines form convex polygons, it runs in $O(n \lg n + s)$ time. Could a data structure such as the territory set of section 1.4 be introduced into the Szymanski–Van Wyk algorithm to allow it to run in worst case time closer to that of the Neivergelt–Preparata algorithm?

The geometric problems discussed in this thesis are only a small number of those problems for which a two–layer model may prove fruitful. Many geometric problems are incorporated into applications programs where the size of the input is arbitrarily large in practice. For example, the convex hull is used in simulating chemical reactions and in estimating population parameters in statistics and triangulation is used in numerical analysis and in computer aided design of VLSI circuits. There are algorithms for these problems in the literature with good or even optimal worst case time bounds, but these algorithms assume all data is available in primary memory at all times. Are there algorithms for these problems based upon the two–layer memory model which match the time bounds of existing algorithms but run in small primary memory and guarentee good paging?

Finally, is there a data structure with the simplicity of interval trees that will allow ennumeration of all k segments that overlap a test interval in time $O(\lg n + k)$ where n is the number of segments in the structure? If not, is there a structure similar to priority search trees that does not have the same level of complexity in insertions and deletions?

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1974.

[2] J. Bentley, D. Haken, and R. Hon, "Statistics on VLSI Design," *Carnegie-Mellon University Department of Computer Science Technical Report No. CMU-CS-80-111*, April, 1980.

[3] J. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE Transactions on Computers*, Vol. C-29, No. 7, July, 1980.

[4] L. Guibas and J. Saxe, "Problem 80-15," *Journal of Algorithms*, Vol. 4, 1983, pp. 177-181.

[5] H. Imai and T. Asano, "Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane," *Journal of Algorithms*, Vol. 4, 1983, pp. 310-323.

[6] E. McCreight, "Priority search trees," *Xerox Corporation Palo Alto Research Center Technical Report*, CSL-81-5, January 1982.

[7] J. Nievergelt and F.P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Communications of the ACM*, Vol. 25, No. 10, October 1982, pp. 739-747.

[8] M. I. Shamos and D. Hoey, "Geometric intersection problems," *Proceedings of the Seventeenth Annual Symposium on Foundations of Computer Science*, IEEE, 1976.

[9] T.A. Standish, *Data Structure Techniques*, Addison-Wesley, 1980.

[10] T. G. Szymanski and C. J. Van Wyk, "Space efficient algorithms for VLSI artwork analysis," *Proceedings of the Twentieth Design Automation Conference*, June 1983, pp. 734-739.

[11] R. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the Association for Computing Machinery*, Vol. 25, No. 2, 1975, pp. 215-225.

[12] J. Vuillemin, "A unifying look at data structures," *Communications of the ACM*, Vol. 23, No. 4, April 1980, pp. 229-239.

END

FILMED

2-86

DTIC